Mechanising proofs of program transformation rules

K. Sere, J. von Wright

RUU-CS-91-11
May 1991

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Mechanising Proofs of Program Transformation Rules*

Kaisa Sere

Department of Computer Science, Åbo Akademi University
P.O. Box 80.085, 3508 TB Utrecht
The Netherlands
e-mail: kaisa@cs.ruu.nl

Joakim von Wright

Swedish School of Economics and Business Education
Biblioteksgatan 16, SF-65100 Vaasa
Finland
e-mail: jvwright@abo.fi

Abstract

In the refinement calculus, programs are developed in a stepwise fashion by repeated application of transformation rules that preserve total correctness. We show how such rules can be formalised and proved using the HOL theorem prover. We also show how hierarchies of rules can be built e.g. by formalising 'generic' rules and how application of these rules can be automated in HOL.

*The work reported here was carried out in the ProCoS II project sponsored by the Science Development Board of Finland (SDBF), the Swedish Board of Technical Development (STU) and the Dutch Foundation for Computer Science (SION). The work was also supported by the Dutch organisation for scientific research under project nr. 52-412 (Specification and Transformation of Programs, STOP).

ISSN: 0924–3275

1 Introduction

Stepwise refinement is a methodology for developing programs from high-level program specifications into efficent implementations. The *refinement calculus* of Back [1,2] is a formalisation of this approach, based on the weakest precondition calculus of Dijkstra [7].

Practical program derivation within the refinement calculus [4] has shown that refinement steps often tend to be very similar to each other. Rather than always inventing a refining statement and proving the correctness of the refinement, it is convenient to have access to a collection of program transformation rules whose correctness has been verified once and for all. Every step in the development process is then an application of such a rule.

General rules are easily formulated in the refinement calculus. The rules are verified within the calculus either by appealing to the definition of correct refinement or to some rule that has already been proved. Collections of rules can thus be designed in hierarchy: correctness proofs of higher-level rules appeal to rules on lower levels.

Correctness proofs of refinement steps and transformation rules involve lots of proof detail and routine work that could in principle be left to a mechanical theorem prover. In this paper we show how hierarchies of transformation rules can be designed and how their proofs can be checked using the theorem prover HOL.

The HOL system (Higher Order Logic, [9]) is a theorem prover, which has been used mainly for formal specification and verification of hardware. However, the higher order features of HOL are convenient also when defining programming notations and semantics and when reasoning about correctness and program transformations.

Our main contribution is the formalisation of a collection of program transformation rules and their hierarchical proofs in HOL. We also show how application of these rules can easily be automated. In this way the result of an application of a transformation rule is mechanically calculated, and hence more trustworthy.

The specific collection of rules that we consider is useful when deriving parallel algorithms as so called *action systems* [3] within the refinement calculus. The method we use to formalise rules and their proofs is, however, completely general and can be used to formalise any set of rules.

The idea of using transformation rules in program development is not new [11]. What we believe is new is our formalisation of their proofs in HOL. Preliminary work in this direction was done in [5]. Recently there have been some attempts to automate the verification of derivations within other frameworks also. It is shown in [10] that the LP theorem prover can be used as a transformation calculator

for program derivation in the Bird-Meertens calculus. An attempt to mechanise UNITY is reported in [8]. The developed proof system is implemented on the Boyer-Moore theorem prover.

Overview We proceed as follows. In Section 2 we present the HOL system. We also describe how the basic concepts of programming, states predicates and predicate transformers, can be formalised in HOL. In Section 3 we show how programs can be derived within the refinement calculus. Program derivation within the refinement calculus relies heavily on the use of program transformation rules. In Section 4 we identify a collection of rules that is useful when deriving parallel programs. We show that such a collection can be built in a hierarchical manner. In Section 5, we show how the correctness of our rules is established within the HOL system and how rule application can be automated. Finally, Section 6 contains some concluding remarks and directions for further research.

2 Using the HOL system

The HOL system is a proof assistant for higher-order logic based on the Edinburgh LCF theorem proving system [9]. This in turn is a combination of predicate calculus and the typed lambda calculus. In this section we describe the main features of the HOL system. We also show how some basic concepts of programming (states, predicates and predicate transformers) can be formalised in HOL.

2.1 Basic concepts of the HOL system

When working with the HOL system one always works inside some *theory*. Within a theory one can define types, constants and axioms and prove theorems. The *HOL theory* (to be distinguished from the HOL system) is a hierarchy of predefined basic theories, defining among other things the types `bool`, representing truth values, and `num`, representing the natural numbers. Together with these go axioms and theorems of logic and arithmetic. Every user-defined theory has the HOL theory as a parent. Thus the definitions, axioms and theorems of the HOL theory are always available.

The logic of HOL is higher-order logic (simple type theory) which was originally developed by Church [6]. The typing means that every entity must have a type assigned to it. Polymorphic types containing type variables are permitted. We write `x:t` to indicate that the object `x` has type `t`. We let type names that begin with an asterisk (`*`) denote type variables.

Types can be combined into *pair types* and *function types*. A pair `(x:ty1,y:ty2)` has type `ty1#ty2`. A function which maps arguments of type `ty1` to values of type `ty2` has type `ty1→ty2`.

Goals (sequents) are pairs `(A,t)`, where `A` is a list of terms (the assumptions) and `t` is a term (the conclusion). If a goal has been proved, a corresponding *theorem* is returned.

It should be noted that the user can define any boolean term (even a contradiction) to be an axiom. A theory where no new axioms are introduced is called *definitional*. The theories described in this paper are definitional. This means that the theories are guaranteed to be consistent.

Standard theories and libraries The HOL system contains a number of standard theories that are parents of every theory. We will use the theories of natural numbers (type `num`) and lists (a list of elements of type `ty` has type `(ty)list`). There are also libraries with theories that can be loaded into any theory. For example, if we want to reason about sets within a theory, we can load a set library.

A note on HOL syntax The HOL systems uses standard ASCII character combinations for the logical symbols. However, in this paper we use ordinary logical syntax (note that the truth values are written `T` and `F` and that quantifiers bind weaker than logical connectives). Application of a function `f` to an argument `x` is written `f x` or `f(x)`. Function application associates to the left so `f x y` is the same as `(f x)y`. We will usually omit parentheses in order to make function expressions more readable. Note that quantifiers bind weaker than logiocal connectives.

Terms of the HOL logic are written in double quotes. Theorems are written with a turnstile symbol (⊢) separating the assumptions list from the conclusion. Lists are written within square brackets with semicolon separating the elements, so `[a;b;c]` is a list with three elements `a`, `b` and `c`. To make example HOL texts more readable, we will generally leave out type information if the types are obvious from the context.

2.2 Proofs in HOL

Proofs in HOL can be made both by forward and backward reasoning. When forward reasoning is used, a valid inference rule is applied to a number of theorems and/or terms, and the result is a new theorem. There are a number of pre-proved inference rules in the HOL system and the user can combine these into new inference rules.

Backward proof is supported by the goal stack of the HOL system. A boolean term can be set up as a goal and then reduced using *tactics*. A tactic is a function which is applied to a goal and returns subgoals. A subgoal which matches an existing theorem is solved. When the backward proof is finished (i.e., when all subgoals have been solved), the HOL system constructs a forward proof of the original goal. HOL tactics can be combined into more powerful tactics by means of special functions called *tacticals*.

The HOL system is embedded in the functional language ML [9]. When starting a HOL session, an ML environment is set up. The user interacts with the HOL system through ML, making definitions and evaluating expressions. The HOL system can be used without much knowledge of ML. However, the more experienced ML programmer can write elaborate tactics that automate large parts of proofs.

2.3 Formalising programming concepts in HOL

We generally consider *states* to have the polymorphic type `*s`. When reasoning about a specific statement, this type will be instantiated to a product type with one component for every (global) program variable. Thus variables have no name; they are identified by their position in the state tuple. In the examples below, we assume that the state has type `bool#num#num`, corresponding to the Pascal-style variable declaration

$$\textbf{var}\ \ b : \mathit{bool};\ x, y : \mathit{num};$$

Predicates and predicate transformers A predicate assigns a truth value to every state. We let `pred` abbreviate the predicate type `*s→bool` (actually, the HOL system does not permit abbreviating a polymorphic type; we do it here for notational convenience only). We will typically use the symbols `p` and `q` for predicates.

We define operators on predicates by lifting the corresponding logical operators. The defining theorems are as follows:

$\vdash_{def}$ `false = λs. F`
$\vdash_{def}$ `true = λs. T`
$\vdash_{def}$ `not p = λs. ¬p s`
$\vdash_{def}$ `p and q = λs. p s ∧ q s`
$\vdash_{def}$ `p or q = λs. p s ∨ q s`
$\vdash_{def}$ `p imp q = λs. p s ⇒ q s`

where `s` has type `*s`. The implication (partial) order on predicates is defined as follows:

$\vdash_{def}$ `p implies q = ∀s. p s ⇒ q s`

Note that `and`, `imp` and `implies` are defined to be infix operators (curried functions).

The predicate $x < y$ is represented as `"λ(b,x,y).x<y"` in our example state space. Note that substitution in predicates is easily achieved by a combination of application and abstraction; e.g., $(x < y)[2/x]$ is formalised as `"λ(b,x,y).(λ(b,x,y).x<y)(b,2,y)"` which is beta reduced to `"λ(b,x,y).2<y"`, exactly as it should.

When reasoning about programs with local variables, we need to consider an enlarged state space of type `*#*s` where the first component is the added variable. Predicates on this state space have type `*#*s->bool`, in this paper abbreviated `epred`.

Predicate transformers have type `pred→pred`, abbreviated `ptrans`, or `epred →epred`, abbreviated `eptrans`. We will typically use the symbol `c` for predicate transformers (the choice is motivated by the fact that we will identify predicate transformers with commands). We then define basic properties of predicate transformers, the two most important of which are monotonicity and conjunctivity:

$\vdash_{def}$ `monotonic c = ∀p q. p implies q ⇒ (c p) implies (c q)`
$\vdash_{def}$ `conjunctive c = ∀p q. c(p and q) = (c p) and (c q)`

We will return to operators on predicate transformers in Section 5.

Properties of predicates and predicate transformers For proving predicate properties, we have devised a tactic `PRED_TAUT_TAC` that proves simple predicate tautologies without quantifiers. It is built on the `TAUT_TAC` tactic from the HOL-library on tautologies. Theorems involving quantification usually require rewriting into boolean form and then standard proof techniques.

We also use least fixpoints of monotonic functions on predicates. We will not go into details here, we simply assume that `fix f` gives the least fixpoint of an arbitrary monotonic function `f` on predicates. The definition of `fix` and the proofs of fixpoint properties are quite straightforward in HOL.

3 Refinement calculus

3.1 Specification language

We consider the language of guarded commands of Dijkstra [7], with some extensions. We have two syntactic categories, statements and actions. A *statement S*

is defined as

$$
\begin{array}{lll}
S ::= & x := e & \textit{(multiple assignment statement)} \\
| & \textit{skip} & \textit{(skip-statement)} \\
| & \textit{abort} & \textit{(abort-statement)} \\
| & \{Q\} & \textit{(assert statement)} \\
| & S_1; S_2 & \textit{(sequential composition)} \\
| & \textbf{if } A_1 \, [\!] \, \ldots \, [\!] \, A_n \textbf{ fi} & \textit{(conditional composition)} \\
| & \textbf{do } A_1 \, [\!] \, \ldots \, [\!] \, A_n \textbf{ od} & \textit{(iterative composition)} \\
| & |[\textbf{var } x; S]| & \textit{(block with local variables)}
\end{array}
$$

Here $A_1, \ldots, A_n$ are actions, g is a boolean expression, x is a (list of) variable(s), e is a (list of) expression(s) and Q is a predicate.

The *assert statement* $\{Q\}$ acts as *skip*, if the condition Q holds in the initial state. If the condition Q does not hold in the initial state, the effect is the same as *abort*. The other statements have their usual meanings.

An *action* (or guarded command) A is of the form $A = g \rightarrow S$ where g is a boolean expression (the *guard* or enabling condition gA) and S is a statement (the *body* sA).

Action systems An action system is a statement of the form

$$|[\textbf{var } x; S_0; \textbf{do } A_1 \, [\!] \, \ldots \, [\!] \, A_n \textbf{ od}]| \tag{1}$$

where x denotes the local variables of the system.

An action system specifies a parallel computation, with non-interfering actions executing in parallel. Action systems and their parallel semantics are discussed in details in [3,4]. The syntactic form (1) of action systems is, however, enough for our purposes.

Weakest preconditions The refinement calculus is based on the *weakest precondition* calculus of Dijkstra [7]. We therefore assume that a weakest precondition semantics is given for the statements. For the assert statement and block with local variables we have that

$$
\begin{aligned}
\mathrm{wp}(\{Q\}, R) &= Q \wedge R \\
\mathrm{wp}(|[\textbf{var } x; S]|, R) &= \forall x . \mathrm{wp}(S, R)
\end{aligned}
$$

For the iteration, we define $\mathrm{wp}(\textbf{do } g \rightarrow S \textbf{ od}, Q)$ to be the least fixpoint of the function

$$(g \vee Q) \wedge (\neg g \vee \mathrm{wp}(S, X))$$

in predicate variable X. With this definition we permit unbounded nondeterminism, which can be introduced e.g., by an uninitialised block.

The weakest preconditions for the other statements are defined in the usual manner.

3.2 Refinement of statements

Let S and S' be statements. Statement S is said to be *refined* by the statement S', denoted $S \leq S'$, if for every postcondition R,

$$\mathrm{wp}(S, R) \Rightarrow \mathrm{wp}(S', R).$$

Refinement captures the notion of statement S' preserving the correctness of statement S. More precisely, $S \leq S'$ holds if and only if $P\langle S\rangle Q \Rightarrow P\langle S'\rangle Q$ for every precondition P and postcondition Q, where $P\langle S\rangle Q$ stands for the total correctness of S w.r.t. P and Q.

We say that the statements S and S' are *(refinement) equivalent*, denoted $S \equiv S'$, if $S \leq S'$ and $S' \leq S$.

The refinement relation is *reflexive* and *transitive*, i.e., it is a preorder. If we identify statements with their weakest precondition predicate transformers (we will do this in Section 5), then the refinement relation is a partial order and refinement equivalence becomes equality.

The statement constructors are also *monotonic* with respect to the refinement relation, i.e.,

$$T \leq T' \Rightarrow S[T] \leq S[T']$$

for any statement S in which T occurs as a substatement ($S = S[T]$). The refinement relation is studied in more detail in [1,2].

Context dependent replacements In practice, we are often faced with a situation where the context S is such that $S[T] \leq S[T']$ does in fact hold, even though $T \leq T'$ does not hold. This kind of *context dependent* replacements can be established correct by the following technique [2].

Assume that we can prove

(1) $S[T] \leq S[\{Q\};\ T]$ (*context introduction*) and

(2) $\{Q\};\ T \leq T'$ (*refinement in context*).

By monotonicity and transitivity we then have that $S[T] \leq S[T']$. The first step introduces information about the context in the form of an assert statement at the appropriate place, the second step uses this information. This shows the importance of assertion statements in the refinement calculus.

Note that we are always permitted to remove any context assertion, i.e.,

$$S[\{Q\};\ T] \leq S[T]$$

is always valid (because $\{Q\};\ S \leq S$ always holds).

Stepwise derivation of programs The refinement relation provides a formalization of the stepwise refinement method for program construction. One starts with an initial high-level specification/program statement S_0, and constructs a sequence of successive refinements of this,

$$S_0 \leq S_1 \leq \ldots \leq S_{n-1} \leq S_n.$$

By transitivity, S_n will then be a refinement of the original program S_0. An individual refinement step may consist of replacing some substatement T of $S_i[T]$ by its refinement T'. The resulting statement $S_{i+1} = S_i[T']$ will then be a refinement of S_i by monotonicity.

The refinement calculus can be also used to derive general program transformation rules. The correctness of the rules is verified once and for all relying on the definition of refinement between two statements or on some other transformation rule that has already been proved. The rules are often accompanied by some application conditions. In practice, a refinement step usually consists of the application of such a transformation rule. The correctness of the refinement step is in this case verified by showing that the application conditions of the rule are satisfied.

In order for the refinement calculus to be really useful in practical program development, the program designer should have access to a library of verified program transformation rules. We will later show how collections of transformation rules can be built up hierarchically within the refinement calculus. The main emphasis is on showing how such a hierarchy can be verified using a mechanical theorem prover, the HOL system.

4 Hierarchies of program transformation rules

In this section, we derive a collection of program transformation rules in a hierarchical manner. The aim of this specific collection is to provide enough rules for turning any initial specification in our language into an action system. Some of the rules are proper refinements while most of them are equivalencies.

4.1 Basic rules

The correctness of rules in this subsection is shown by an application of the definition of correct refinement between statements, i.e., by appealing to the weakest precondition semantics of the statements (we omit the rather straightforward proofs of these rules here).

Assert statement The rules for the assert statement allow us to introduce (and remove) context information in a program text. These rules are very important in practice: they make it possible to do context dependent replacements. Also most of the other rules assume that some context assertions are present. Such assertions can be introduced using these rules.

The first two rules show how assertions can be introduced and removed.

$$\mathrm{wp}(S, Q) = \mathit{true} \quad \Rightarrow \quad S \equiv S; \{Q\} \tag{2}$$

$$\{Q\}; S \leq S. \tag{3}$$

An assert statement is refined by another assert statement provided that the refining assertion is weaker than the original assertion:

$$(P \Rightarrow Q) \Rightarrow \{P\} \leq \{Q\} \tag{4}$$

Assertions can be propagated forwards and backwards in a program text:

$$P \Rightarrow \mathrm{wp}(S, Q) \quad \Rightarrow \quad \{P\}; S \leq S; \{Q\} \tag{5}$$

$$S; \{Q\} \equiv \{\mathrm{wp}(S, Q)\}; S; \{Q\} \tag{6}$$

There are many other rules for manipulating and propagating assertions. For details, we refer to [2,4].

if-introduction A sequentially composed context assertion and arbitrary statement can be replaced with an if-statement that contains a single action only.

$$\{g\}; S \equiv \textbf{if } g \rightarrow S \textbf{ fi}. \tag{7}$$

Fold-unfold Our next rule is a basic loop folding and unfolding rule.

$$\textbf{do } g \rightarrow S \textbf{ od} \equiv \textbf{if } g \rightarrow S; \textbf{do } g \rightarrow S \textbf{ od} \,[\!]\, \neg g \rightarrow \mathit{skip} \textbf{ fi}. \tag{8}$$

Manipulating actions The following two rules apply for both conditional composition and iterative composition although we give them only for iteration.

$$\textbf{do } A_1 \parallel A_2 \textbf{ od} \equiv \textbf{do } A_2 \parallel A_1 \textbf{ od} \tag{9}$$

$$\textbf{do } A_1 \parallel A_2 \textbf{ od} \equiv \textbf{do } gA_1 \vee gA_2 \rightarrow \textbf{if } A_1 \parallel A_2 \textbf{ fi od}. \tag{10}$$

We can always eliminate disabled actions from conditional composition and iteration:

$$\{\neg gA_1\}; \textbf{if } A_1 \parallel A_2 \textbf{ fi} \equiv \{\neg gA_1\}; \textbf{if } A_2 \textbf{ fi} \tag{11}$$

$$\{\neg g_1\}; \textbf{do } g_1 \rightarrow S_1 \parallel g_2 \rightarrow S_2; \{\neg g_1\} \textbf{ od} \equiv \{\neg g_1\}; \textbf{do } g_2 \rightarrow S_2; \{\neg g_1\} \textbf{ od}. \tag{12}$$

As a special case of (12) we have a rule for do-elimination:

$$\{\neg g\}; \textbf{do } g \rightarrow S \textbf{ od} \equiv \{\neg g\}. \tag{13}$$

Introducing local variables Using the following rule we can add new local variables and assignments to these into arbitrary statements.

$$S \equiv |[\ \textbf{var } x; S[x := h/skip]\ ]|. \tag{14}$$

The rule holds provided that S is a statement that does not contain any occurrence of the variable x. This rule is very important when designing parallel algorithms as we often need to model the replication of data among a collection of processes e.g. when we change a scalar variable into an array for loop parallelisation. This is done in the form of new auxiliary variables, see [4].

4.2 Derived rules

More powerful rules can be derived based on the basic rules. The correctness proofs of these rules appeal to rules that have already been proved. The rules below are used in practice to derive action systems from other statements in the specification language.

Restricted fold-unfold The first rule is a loop fold-unfold rule that can be applied only in a certain context:

$$\{g\}; \textbf{do } g \rightarrow S \textbf{ od} \equiv \{g\}; S; \textbf{do } g \rightarrow S \textbf{ od}. \tag{15}$$

This folding rule is useful as such, but it also makes the subsequent proofs easier to carry out. The correctness of this rule is established by appealing to the basic rules (8), (9), (11) and (7).

Creating action systems Finally, we give rules to construct iterative compositions. These rules do not make much sense in strictly sequential programs, but become important, when deriving action systems, as iterative compositions are the basic ingrediences in these systems.

An application of the first rule turns any statement into an iterative composition.

$$\{g\}; S; \{\neg g\} \;\equiv\; \{g\}; \textbf{do } g \to S; \{\neg g\} \textbf{ od}. \tag{16}$$

We give the proof of this rule below. The same proof is given later in the HOL framework for comparison.

$$\begin{array}{lll} & \{g\}; \textbf{do } g \to S; \{\neg g\} \textbf{ od} & \\ \equiv & \{g\}; S; \{\neg g\}; \textbf{do } g \to S; \{\neg g\} \textbf{ od} & \{ \text{ by (15)} \} \\ \equiv & \{g\}; S; \{\neg g\} & \{ \text{ by (13)} \} \end{array}$$

Using the following rule, a statement can be turned in to an action and moved into an iterative composition.

$$\{Q\}; S; \{\neg g\}; \textbf{do } g' \to S'; \{\neg g\} \textbf{ od} \;\equiv\; \{Q\}; \textbf{do } g \to S; \{\neg g\} \;[\!]\; g' \to S'; \{\neg g\} \textbf{ od} \tag{17}$$

where $Q = g \wedge \neg g'$. Repeated application of rules (16) and (17) makes it possible to create an action system with more than two actions from a sequential composition of statements.

An if-statement can be turned into an iterative composition provided that the single action created disables itself.

$$\textbf{if } g \to S; \{\neg g\} \textbf{ fi} \;\equiv\; \{g\}; \textbf{do } g \to S; \{\neg g\} \textbf{ od}. \tag{18}$$

5 Proving transformation rules in HOL

We shall now show how action systems can be formalised in HOL and how transformation rules are proved and applied in the HOL system. In the HOL formalisation, we identify statements with their weakest precondition predicate transformers. To distinguish between statements and their HOL formalisations, we shall call the latter *commands*. A command `c` will thus have type `ptran` and we write `c q` rather than `wp(c,q)` for the weakest precondition of command `c` with respect to predicate `q`. Actions are formalised as pairs `(g:pred,c:ptrans)` where `g` is the guard and `c` is the command.

5.1 Definition of action system commands

To keep as close to the syntax of action systems as possible we give names to the commands that occur in action systems. For conditional composition and iteration we first define restricted versions. We have the following defining theorems (where v is the state tuple and q has type pred):

```
⊢def assign e q =  λv. q (e v)
⊢def skip q = q
⊢def abort q = false
⊢def assert p q = p and q
⊢def (c1 seq c2) q = c1(c2 q)
⊢def if2 (g1,c1) (g2,c2) q = (g1 or g2) and (g1 imp (c1 q)) and (g2 imp (c2 q))
⊢def do1 (b,c) q =  fix(λp. (b or q) and ((not b) or (c p)))
⊢def block (c:eptrans) q = λv. ∀x. c (λ(x,v). q v)(x,v)
```

Note that the argument e of the assignment is a *state expression*, i.e., a n-valued function (where n is the number of state components) of type *s→*s and that seq is defined to be an infix operator. Note also that the command c in the block is a predicate transformer on an extended state space. The local variable x has the polymorphic type *.

A command that is formed using the action system constructs defined above will be called an *action system command.* As an example command on the state space type bool#num#num (corresponding to a variable triple (b, x, y)), we consider the multiple assignment $x, y := y+1, x$. It is formalised in HOL as "assign λ(b,x,y).(b,y+1,x)".

General conditionals and iterations Using list recursion we can define conditionals and iterations with an arbitrary number of actions (we first define lguard of an action list to be the disjunction of the guards in the list):

```
⊢def (lguard [] = false) ∧ (lguard (CONS a al) = (FST a) or (lguard al))
⊢def (if [] = abort) ∧ (if (CONS a al) = if2 a (lguard al,if al))
⊢def do al = do1 (lguard al,if al)
```

with action a:pred#ptrans and action list al:(pred#ptrans)list. In HOL syntax, [] is the empty list and FST and SND are pair projections.

For convenience, we also define one-action conditional and two-action iteration:

```
⊢def if1 (g,c) = if [(g,c)]
⊢def do2 (g1,c1) (g2,c2) = do [(g1,c1);(g2,c2)]
```

5.2 Proving basic properties of commands

When we work with action systems, the restricted syntax permits us to assume that all statements are monotonic and conjunctive.

In our formalisation we work with predicate transformers, and we cannot assume that monotonicity and conjunctivity always hold. Thus, some program transformation rules in HOL will have explicit monotonicity or conjunctivity assumptions. To automate the proof that a given action system command is monotonic or conjunctive, we have then written conversions (a conversion is an ML function which takes a term as an argument and returns a theorem) `prove_mono` and `prove_conj`.

An example of a theorem that is usually taken for granted but must be proved explicitly in the HOL formalisation is the associativity of sequential composition:

```
⊢ (c1 seq c2) seq c3 = c1 seq (c2 seq c3)
```

The action system command syntax contains some redundancy. Also, simple cases of the `if` and `do` commands can be rewritten using other commands. The following theorems are examples of such relations between commands (the proofs are simple exercises in using HOL).

```
⊢ abort = assert false
⊢ skip = assert true
⊢ do [(g,c)] = do1 (g,c)
```

The refinement relation The refinement relation `ref` is defined as an infix operator:

```
⊢def c ref c' = ∀q. (c q) implies (c' q)
```

The fact that `ref` is a partial order on commands order follows immediately from the fact that `implies` is a partial order on predicates. Thus refinement equivalence is represented by equality.

5.3 Proving refinement rules

We next prove the correctness of program transformation rules using the ideas of Section 4. Thus basic rules are proved by appealing to the semantics. After that higher-level rules can be proved by means of the basic rules and so on. As an example, we show a proof dialogue for one simple higher-level rule.

Proving basic rules We shall now discuss how the basic rules shown in Section 4 can be proved in our HOL-formalisation. We will not show the proofs in any detail. Rather we want to describe how the proofs can be performed.

The basic rules (2)–(6) for assertions are easily proved by rewriting with the definitions and then applying the predicate tautology proving tactic PRED_TAUT_TAC:

```
conjunctive c ⊢ (c q = true) ⇒  (c = c seq (assert q))
⊢ ((assert q) seq c) ref c
⊢ p implies q ⇒  (assert p) ref (assert q)
conjunctive c ⊢ p implies (c q) ⇒
                    ((assert p) seq c = (assert p) seq c seq (assert q))
⊢ c seq (assert p) = (assert (c p)) seq c seq (assert p)
```

(note the conjunctivity assumptions in two of the rules).

The rules for if-introduction (7) and ∨-distributivity (10) follow directly from the definitions:

```
⊢ (assert g) seq c = if1 (g,c)
⊢ do2 (g1,c1) (g2,c2) = do1 (g1 or g2, if2 (g1,c1) (g2,c2))
```

The rule for eliminating a loop (13) and the rule for unfolding (8),

```
⊢ (assert (not g)) seq do1(g,c) = assert (not g)
monotonic c ⊢ do1 (g,c) =  if2 (g,c seq (do1 (g,c))) (not g,skip)
```

both follow from the fixpoint definition of do. The rule for eliminating a disabled action in a loop (12),

```
monotonic c ⊢ (assert (not g1)) seq (do2 (g1,c1) (g2,c2 seq (assert (not g1))))
                  = (assert (not g1)) seq (do1 (g2,c2 seq (assert (not g1))))
```

is a bit harder to prove. The proof is divided into two refinement proofs, both using the following two lemmas (where DO stands for the statement **do** $g \rightarrow S$ **od** and S is assumed to be monotonic):

$$\begin{array}{rcl} (\forall Q.\ \mathrm{wp}(S,Q) \Rightarrow \neg P \vee \mathrm{wp}(S',Q)) & \Rightarrow & \{P\};S \leq \{P\};S' \\ ((g \vee Q) \wedge (\neg g \vee \mathrm{wp}(S,X)) \Rightarrow X) & \Rightarrow & (\mathrm{wp}(DO,Q) \Rightarrow X) \end{array}$$

The first of these lemmas is proved from definitions and the second one from the defining properties of least fixpoints.

Proving higher-level rules Having proved the basic rules we can prove higher-level rules much in the same way as we would prove them by hand, i.e., as a sequence of rewriting steps where every step appeals to a previously proved rule.

As an example, we shall show the proof of the rule for turning a statement into an action system (16).

To do this, we assume that we have already proved the restricted fold-unfold rule (15),

```
monotonic c ⊢ (assert g) seq (do1(g,c)) = (assert g) seq c seq (do1(g,c))
```

In our formalisation, the rule for turning a statement into an action system (16) is expressed in the following theorem:

```
monotonic c ⊢ (assert g) seq c seq (assert(not g)) =
                  (assert g) seq do1(g,c seq (assert (not g)))
```

The HOL proof of this theorem follows the proof shown in Section 4.2 quite closely. However, there are two additions. We need to instantiate `c` to `c seq (assert (not g))` in the previously proved restricted fold-unfold rule before we can use it in the proof. We also rely on the associativity of sequential composition.

In the boxed HOL dialogues below, lines beginning with the prompt character `#` are user input (terminated with `;;`), while the other lines are HOL replies (for brevity, we show only the most important reply lines). Tactics are applied to the goal through the ML function **expand**. We use two predefined tactics, `IMP_RES_TAC` which adds assumption using resolution, and `REWRITE_TAC` which rewrites using theorems in the argument list as well as some standard rewriting theorems.

The proof proceeds as follows. First the goal is set up (using the function `set_goal`).

```
#set_goal(["monotonic c"],
#      "(assert g) seq c seq (assert(not g)) =
#          (assert g) seq do1(g,c seq (assert (not g)))");;
"(assert g) seq (c seq (assert(not g))) =
 (assert g) seq (do1(g,c seq (assert(not g))))"
    [ "monotonic c" ]
```

We next use the following monotonicity lemma `lemma1` to add a resolved assumption:

```
⊢ monotonic c ⇒  monotonic (c seq (assert (not g)))
```

(the proof of this lemma is straightforward).

```
#expand(IMP_RES_TAC lemma1);;
OK..
"(assert g) seq c seq (assert(not g)) =
  (assert g) seq do1(g,c seq (assert (not g)))"
    [ "monotonic c" ]
    [ "monotonic(c seq (assert(not g)))" ]
```

Now the goal is rewritten using `lemma2`, which is the restricted fold-unfold rule (15) with `c` instantiatiated to `c seq (assert(not g))`.

```
#expand(REWRITE_TAC[lemma2]);;
OK..
"(assert g) seq (c seq (assert(not g))) =
 (assert g) seq
 ((c seq (assert(not g))) seq (do1(g,c seq (assert(not g)))))"
    [ "monotonic c" ]
    [ "monotonic (c seq (assert(not g)))" ]
```

Finally we rewrite using associativity of `seq` (which has been proved in the theorem `seq_assoc`) and do-elimination (13). This solves the goal and the HOL system constructs the theorem corresponding to the initial goal (the HOL system prints the assumption of the theorem as a dot).

```
#expand(REWRITE_TAC[seq_assoc;do-elimination])
OK..
goal proved
. ⊢ (assert g) seq (c seq (assert(not g))) =
    (assert g) seq (do1(g,c seq (assert(not g))))
```

The rule for turning an if-statement into an action system is proved directly from the previous rule. The rule for adding a statement into a loop is proved in a similar way, although the proof is much longer.

The rule for introducing local variables The rule for introducing a local variable (14) cannot be proved as a single theorem in our theory. There are many reasons for this. First, the two occurrences of S in the rule are (in our formalisation) two separate objects, as they have distinct types (`ptrans` and `eptrans`). Second, we cannot describe an assignment as "assigning only to the variable x". Finally, we cannot formulate the side condition "S does not contain any occurence of x".

Instead of proving a single theorem corresponding to the rule, we can write an ML conversion that proves any instance of the theorem. Such a conversion takes two arguments: the original command `c` and the transformed command `block c'` (where `c'` is related to `c` as $S[x := h/skip]$ is related to S) and then returns a theorem stating that they are equal.

5.4 Applying rules using HOL

We have shown how to prove the correctness of program transformation rules using HOL. We now consider applying the rules to an actual command (i.e., a term representing a program text).

To simplify application of a specific rule to a term, we write a conversion that takes the term as one argument and possibly a number of other arguments (depending on the rule) and returns the desired refinement as a theorem. As a trivial example, we show how to do the following assertion introduction:

$$x := 1 \quad \equiv \quad x := 1; \{x = 1\}$$

when working in the two-variable (x, y) state space of type `num#num`.

We take the rule for introducing an assertion (2):

```
conjunctive c ⊢ (c q = true) ⇒  (c = c seq (assert q))
```

The conversion for this rule is called `apply_assert_intro`. It takes the original command `c` (which may be a complicated expression) as a first argument and the desired post-assertion `q` as a second argument. It then proves that the first argument is a conjunctive command (using `prove_conj` described above) and instantiates the rule. The dialogues below show what this looks like in practice. We first let `t1` represent the command $x := 1$ and `t2` represent the predicate $x = 1$. Then we perform the rule application.

```
#let t1 = "assign λ(x,y).(1,y)";;
t1 = "assign λ(x,y).(1,y)":term

#let t2 = "λ(x,y).(x=1)";;
t2 = "λ(x,y).(x=1)":term

# let th1 = apply_assert_intro t1 t2;;
th1 = ⊢ (assign λ(x,y).(1,y) (λ(x,y).(x=1)) = true) ⇒
            (assign λ(x,y).(1,y)) =
                        (assign λ(x,y).(1,y)) seq (assert λ(x,y).(x=1))
```

We still have a proof obligation left: the antecedent of the implication in the theorem `th1`. Assuming that it is proved in theorem `th2` (that proof is a simple HOL exercise), we can get the desired result using modus ponens (the HOL inference rule `MP`):

```
# let th3 = MP th1 th2;;
th3 = ⊢ (assign λ(x,y).(1,y)) =
            (assign λ(x,y).(1,y)) seq (assert λ(x,y).(x=1))
```

Inspection shows that `th3` formalises the desired refinement equivalence.

In a more user-friendly program development environment, the proof obligation should be presented to the user. After the user has indicated how it is to be proved, the system should automatically perform the modus ponens inference and produce the theorem `th3`.

Refining a subcomponent In general, we want to refine only a small subcomponent T of a large program text. If the subcomponent is replaced by an equivalent component T' then the subcomponent replacement is a simple case of substituting equals in HOL theorems. It is possible to write a conversion that performs the subcomponent replacement in the case when T is strictly refined by T'. This conversion takes as arguments a term representing a program $\lambda X.S(X)$ with the subcomponent indicated by the dummy X and a theorem representing the refinement $T \leq T'$ in isolation. It returns the theorem representing the whole refinement $S[T] \leq S[T']$. For details of how this is implemented, we refer to [5].

6 Conclusion

We have shown how program transformation rules used in the action system approach to program development can be proved using the HOL system. Our formalisation of action systems is based on previous work in [5], but we have adopted a more realistic formalisation of the state space. This formalisation permits local variables to be handled in a satisfactory way.

By designing a hierarchy where higher-level rules are proved by appealing to lower-level rules, we can prove the correctness of powerful rules while keeping the proofs reasonably short. The HOL system supports this hierarchical approach. Proving the transformation rules using HOL gives us confidence in the correctness of these rules. This is especially valuable in the case of transformation rules for loops, where the formulas involved can be quite complicated and where intuitive reasoning may lead to false conclusions.

Once a transformation rule has been proved correct, it can be applied to a given action system, possibly with the restriction that some side conditions must be satisfied. We have shown that one can write ML functions which perform rule application. This way a refinement step is implemented as a theorem that has been proved automatically in HOL.

A lot remains to be done before the building blocks described in this paper can be used in serious program development work. First of all, a more comprehensive rule collection has to be designed and proved in HOL. Flexible automatic

[10] U. Martin and T. Nipkow. Automating squiggol. In M Broy, editor, *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 223–236. April 2–5 1990. Sea Gallilee, Israel.

[11] H. Partsch and R. Steinbrügge. Program transformation systems. *ACM Computing Surveys*, 15:199–236, 1983.